SPADE: The System S Declarative Stream Processing Engine

Buğra Gedik
IBM Thomas J. Watson Research Center, Hawthorne, NY, 10532, USA
bgedik@us.ibm.com

Henrique Andrade
IBM Thomas J. Watson Research Center, Hawthorne, NY, 10532, USA
hcma@us.ibm.com

Kun-Lung Wu
IBM Thomas J. Watson Research Center, Hawthorne, NY, 10532, USA
klwu@us.ibm.com

Philip S. Yu
Department of Computer Science, University of Illinois, Chicago, IL, 60607, USA
psyu@cs.uic.edu

MyungCheol Doo
College of Computing, Georgia Institute of Technology, GA, 30332, USA
mcdoo@cc.gatech.edu

ABSTRACT

In this paper, we present SPADE − the System S declarative stream processing engine. System S is a large-scale, distributed data stream processing middleware under development at IBM T. J. Watson Research Center. As a front-end for rapid application development for System S, SPADE provides (1) an intermediate language for flexible composition of parallel and distributed data-flow graphs, (2) a toolkit of type-generic, built-in stream processing operators, that support scalar as well as vectorized processing and can seamlessly inter-operate with user-defined operators, and (3) a rich set of stream adapters to ingest/publish data from/to outside sources. More importantly, SPADE automatically brings performance optimization and scalability to System S applications. To that end, SPADE employs a code generation framework to create highly-optimized applications that run natively on the Stream Processing Core (SPC), the execution and communication substrate of System S, and take full advantage of other System S services. SPADE allows developers to construct their applications with fine granular stream operators without worrying about the performance implications that might exist, even in a distributed system. SPADE's optimizing compiler automatically maps applications into appropriately sized execution units in order to minimize communication overhead, while at the same time exploiting available parallelism. By virtue of the scalability of the System S runtime and SPADE's effective code generation and optimization, we can scale applications to a large number of nodes. Currently, we can run SPADE jobs on ≈ 500 processors within more than 100 physical nodes in a tightly connected cluster environment. SPADE has been in use at IBM Research to create real-world streaming applications, ranging from monitoring financial market feeds to radio telescopes to semiconductor fabrication lines.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Distributed databases*; H.2.3 [**Database Management**]: Languages—*Data manipulation languages*

General Terms

Design

Keywords

Distributed Data Stream Processing

1. INTRODUCTION

On-line information sources are increasingly taking the form of *data streams*, that is time ordered series of events or readings. Example data streams include live stock and option trading feeds in financial services, physical link statistics in networking and telecommunications, sensor readings in environmental monitoring and emergency response, and satellite and live experimental data in scientific computing. The proliferation of these sources has created a paradigm shift in how we process data, moving away from the traditional "store and then process" model of database management systems toward the "on-the-fly processing" model of emerging data stream processing systems (DSPSs). This paradigm shift has recently created a strong interest in DSPS-related research, in academia [1, 4, 6, 7] and industry [8, 17, 21, 25] alike.

In this paper we describe the design of SPADE, which is the declarative stream processing engine of the massively scalable and distributed System S − a large-scale stream processing middleware under development at IBM Research. SPADE provides a rapid application development front-end for System S. Concretely, SPADE offers:

1. *An intermediate language for flexible composition of parallel and distributed data-flow graphs.* This language sits in-between higher level programming tools and languages such as the System S IDE or stream

SQL[1] and the lower level System S programming APIs. The SPADE language provides constructs such as loops, stream bundles, node pools, and operator partitions to ease the specification and configuration of flow graphs in various distributed environments.

2. *A toolkit of type-generic built-in stream processing operators.* SPADE supports all basic stream-relational operators with rich windowing and punctuation semantics. It also seamlessly integrates built-in operators with user-defined ones. One particularly powerful feature of built-in SPADE operators is their ability to operate on *list types* and their ability to intermix scalar and vectorized processing on lists.

3. *A broad range of stream adapters.* These adapters are used to ingest data from outside sources and publish data to outside destinations, such as network sockets, relational and XML databases, file systems, etc.

SPADE leverages the existing stream processing infrastructure offered by the Stream Processing Core (SPC) [2] component of System S. Given an application specification in SPADE's intermediate language, the SPADE compiler generates optimized code that runs on SPC as a native System S application. As a result of this code generation framework, SPADE applications enjoy a variety of services provided by the System S runtime, such as placement and scheduling, distributed job management, failure-recovery, and security. More importantly, this multi-layered framework creates opportunities for the SPADE compiler to perform various optimizations, so as to best map the higher level SPADE constructs into the lower-level ones that the System S runtime expects in order to efficiently run a distributed stream processing application. For instance, SPADE enables developers to structure their applications using fine granular stream operators without worrying about the performance implications that might exist in a distributed system. SPADE's optimizing compiler automatically maps applications into appropriately sized execution units in order to minimize the communication overhead, while at the same time exploiting available parallelism.

SPADE's effective code generation and optimization framework enables it to fully exploit the performance and scalability of System S. It currently runs on approximately 500 processors within more than 100 physical nodes in a tightly connected cluster environment. SPADE has been in use at IBM Research to create real-world data stream processing applications, ranging from processing financial market feeds to radio telescopes to semiconductor fabrication lines.

In summary, SPADE improves over the current state-of-the-art in the following aspects:

- *Sheer scale and performance*: SPADE inherits its scalability from the System S Stream Processing Core, and provides both language constructs and compiler optimizations to fully utilize and expose the performance and flexibility of SPC. The distributed flow-graph composition constructs of SPADE offer an easy way to harness the power of System S, whereas the compiler optimizations deliver high-performance stream processing operators, which can be ideally partitioned into properly sized execution units to best match the runtime resources of System S.

- *Incremental application composition and deployment*: SPADE applications are expected to be long-running continuous queries. These applications can be developed and deployed incrementally. In other words, a deployable application component (a SPADE job/query) can tap into existing streams that are generated by already deployed SPADE or non-SPADE System S jobs. Such connections can optionally be determined dynamically at run-time, using SPC's ability to *discover* source streams based on type compatibility.

- *Relational, non-relational, and mixed workloads*: SPADE supports all fundamental stream-relational operators, with extensions to process list types. Supporting list types and vectorized operations on them enables SPADE to handle, without performance penalty, mixed-workloads, such as those in signal processing applications that usually treat a list of samples as the basic unit of data processing (see SigSegs [12]).

The rest of this paper is organized as follows. Section 2 gives the architectural overview of SPADE and relevant aspects of System S. Section 3 describes the SPADE language and operators. Section 4 discusses compiler optimization opportunities within SPADE's code generation framework. Section 5 introduces SPADE's optimizing partitioner. Section 6 showcases an example SPADE application from the finance engineering domain. Section 7 reports our ongoing work and future directions. Finally, Section 8 concludes the paper.

2. SYSTEM OVERVIEW

In this section we briefly describe System S and provide relevant details of the SPC runtime components utilized by SPADE. We conclude with an overview of SPADE's code-generation framework.

2.1 System S Overview

System S is a large-scale distributed data stream processing middleware. It supports structured as well as unstructured data stream processing and can be scaled from one to thousands of compute nodes. System S runtime can execute a large number of long-running jobs (queries) that take the form of *Data-Flow Graph*s. A data-flow graph consists of a set of *Processing Elements* (PEs) connected by streams, where each stream carries a series of *Stream Data Objects* (SDOs). The PEs implement data stream analytics and are basic execution units that are distributed over the compute nodes. The PEs communicate with each other via their input and output ports, connected by streams. The PE ports as well as streams connecting them are typed. System S adopts the UIMA framework [22] for the type system. PEs can be explicitly connected using hard-coded links (e.g., input port 0 of PE A is connected to output port 1 of PE B) or through implicit links that rely on type compatibility (e.g., input port 0 of PE A is connected to any output port that provides a superset of what it expects). The latter type of connections is dynamic and allows System S to support incremental application development and deployment. Besides these fundamental functionalities, System S provides several other services, such as reliability, scheduling

[1]At the time of this writing, the support for stream SQL on System S is still under development and not yet available.

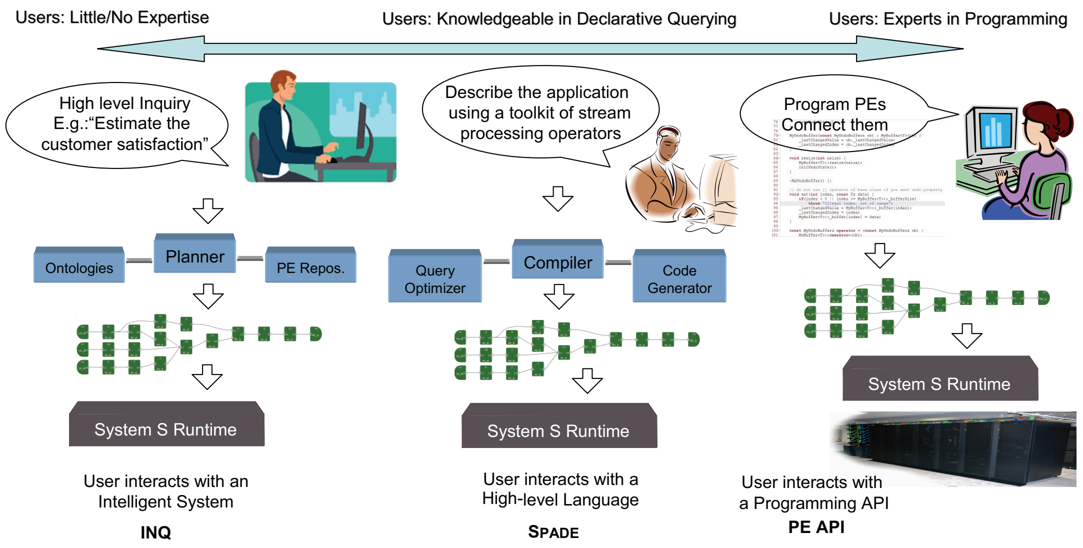
Users: Little/No Expertise
Users: Knowledgeable in Declarative Querying
Users: Experts in Programming
High level Inquiry E.g.:"Estimate the customer satisfaction"
Describe the application using a toolkit of stream processing operators
Program PEs Connect them
Ontologies
Planner
PE Repos.
Query Optimizer
Compiler
Code Generator
System S Runtime
System S Runtime
System S Runtime
User interacts with an Intelligent System
INQ
User interacts with a High-level Language
SPADE
User interacts with a Programming API
PE API

Figure 1: System S from an application developer's perspective

and placement optimization, distributed job management, storage services, and security, to name a few.

System S provides several alternatives for a user or developer to craft data-flow graphs, as shown in Figure 1. At one extreme, an experienced developer can use a programming language such as C++ or Java to implement the desired stream analytics as PEs, utilizing System S' PE APIs. In this case, the developer also creates PE templates that describe each PE in terms of its input and output ports, and populates a configuration file that describes the topology of the data-flow graph. These activities could be simplified via the use of the System S IDE.

At the other extreme, a user with little or no expertise could pose natural language-like, domain-specific *inquiries* to the system. The *Inquiry Services* (INQ) planner can use an existing set of PEs developed for the particular domain at hand, together with their semantic descriptions and a domain ontology, to automatically create a data-flow graph that implements the user's high-level inquiry. For further details, we refer the reader to [19].

In contrast, SPADE strikes a middle-ground between the aforementioned two alternatives, by providing a declarative processing front-end to the users, while still making it possible to integrate arbitrary user-defined or legacy code into the data-flow graph. Developers interacting with SPADE use a set of well-defined, type-generic, and highly configurable operators and stream adapters to compose their applications. SPADE's intermediate language also provides several constructs to ease the development of distributed data-flow graphs, and exposes various knobs to influence their deployment. Furthermore, it forms a common ground on top of which support for other interfaces can be build. For instance, the INQ planner can potentially generate SPADE applications from high-level inquiries, or a StreamSQL query specification can be converted into a SPADE application.

2.2 Stream Processing Core Runtime

Since SPC provides the execution and communication substrate for SPADE, the basics of how a data-flow graph is executed by the runtime is important in understanding SPADE's code generation and optimization framework. Figure 2 shows the key architectural components of SPC runtime.

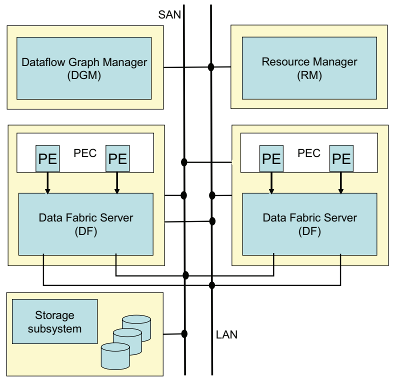

Figure 2: Stream Processing Core (parts relevant to Spade are shown)

The *Dataflow Graph Manager* (DGM) determines stream connections among PEs, and matches stream descriptions of output ports with the flow specifications of input ports. The *Data Fabric* (DF) is the distributed data transport component, comprising a set of daemons, one on each node supporting the system. Upon guidance from the DGM, it establishes the transport connections between PEs and moves SDOs from producer PEs to consumer PEs. The *Resource Manager* (RM) collects runtime statistics from the DF daemons and the PE Execution Containers. This information is used by the System S optimizer, a component called SODA, for making global placement and scheduling decisions. The *PE Execution Container* (PEC) provides a runtime context and access to the System S middleware. It acts as a security barrier, preventing the user-written applications from corrupting the System S middleware as well as each other. For further details on the SPC the reader is referred to [2].

2.3 SPADE's Code Generation Framework

Developers interact with SPADE through SPADE's intermediate language and the SPADE compiler. The SPADE compiler takes a query (job) specification in SPADE's intermediate language as input and generates all the artifacts commonly associated with a native System S application. Figure 3 illustrates the details of this process. The SPADE compiler first generates code that implements the stream operator instances specified in the SPADE query, and then generates additional code to pack these operators into PEs that form the basic execution units distributable over a System S cluster. This mapping can be optimized manually (by the user through language constructs) or automatically (by the compiler through learning, see Section 4). The compiler also generates PE templates, a type system specification, a PE topology that describes the connections among PEs and PE-to-node assignments, and node pools that list the compute nodes to be used during execution. These are fed into the System S Job Description Language (JDL) compiler to yield a complete job description. The operator and PE code are compiled into executable binaries, using traditional programming language compilers and linking against the SPADE and other System S libraries. The JDL file and the set of PE binaries form a readily deployable job on a System S cluster running the Stream Processing Core.

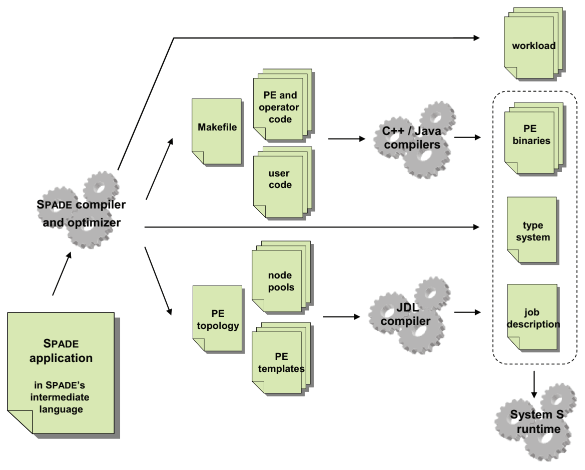

Figure 3: Spade's code generation framework

In order to support customizable data stream processing operators, SPADE relies on a code generation framework, instead of having type-generic operator implementations that employ some form of condition interpretation and type introspection. The reliance on code generation provides the means for the creation of highly optimized platform- and application-specific code. In contrast to traditional database query compilers, the SPADE compiler outputs code that is very tailored to the application at hand as well as system-specific aspects such as: the underlying network topology, the distributed processing topology for the application (i.e., where each piece will run), and the computational environment, including hardware and architecture-specific tuning. In most cases, applications created with SPADE are long-running queries. Hence the long runtimes amortize the build costs. Nevertheless, the SPADE compiler has numerous features to support incremental builds as the application gets modified, greatly reducing the build costs as well.

3. PROGRAMMING MODEL

The SPADE programming model consists of a programming language and the ancillary support runtime libraries and tooling (e.g., parser, code generators, and optimizer). The programming model was conceived with several goals in mind. On the one hand, we aimed at providing high-level constructs where application and tool writers alike can quickly assemble their applications. On the other hand, we focused on creating a framework that enables the compiler to have direct access to the important optimization knobs such that applications are able to derive the best performance from the underlying runtime system. A longer discussion on these topics will come later. Finally, the programming model was conceived such that the out-of-the-box constructs can be extended by adding new language operators and by extending the existing language operators with new capabilities. Specifically, we designed the programming model as well as the tooling to support the addition of external *edge adapters* as well as new *operators*, enabling developers to incrementally add additional operators, forming new, and potentially shareable, *toolkits*.

3.1 Guiding Principles

We believe that two particular design decisions we made were fundamental in achieving the goals stated above: (1) a stream-centric design, and (2) an operator-level programming model. The stream-centric design implies building a programming language where the basic building block is a stream. In other words, an application writer can quickly translate the flows of data she anticipates from a back-of-the-envelope prototype into the application skeleton, by simply listing the data flows. The second aspect, i.e., operator-level programming, is focused on designing the application by reasoning about the *smallest* possible building blocks that are necessary to deliver the computation an application is supposed to perform. Here it is important to note that, while it is hard to precisely define what an operator is, in most application domains, application engineers typically have a good understanding about the collection of operators they intend to use. For example, database engineers typically conceive their applications in terms of the operators available in the stream relational algebra [5, 23]. Likewise, MATLAB [20] programmers have several toolkits at their disposal, from numerical optimization to symbolic manipulation to signal processing, which, depending on the application domain, are appropriately used.

The importance of an operator-centric view of applications is two fold. On one hand, it gently nudges application writers to *think* in terms of fine-granularity operations, that is, the fundamental processing pieces that need to be put together. On the other hand, it exposes multiple optimization opportunities (namely, the *inner-workings* of the operator as well as the operator boundaries) that are important for generating distributed (and parallel) code that will, ultimately, run efficiently on the computing resources. Note that a side benefit of this approach is that, through a recompilation, one can typically obtain different versions of the same application which are specifically optimized for different computational platforms. For example, the runtime application layout as well as the internal operator implementation for a cluster of x86 nodes may not necessarily be the same as the one for a large Symmetric Multiprocessor box. The SPADE code generators were designed with such specializations in mind.

3.2 The SPADE Programming Language

From the programming standpoint, SPADE's syntax and structure is centered on exposing the controls to the main tasks associated with designing applications. At the same time, it effectively hides the complexities associated with: (1) basic data streaming manipulations (e.g., generic language support for data types and building block operations); (2) application decomposition in a distributed computing environment (e.g., how should the application be laid out in the computing environment); and (3) the underlying computing infrastructure and data transport issues (e.g., where to deploy each operator, how to best ingest external data and externalize the data being produced, etc). Many of these aspects can be seen in the SPADE source code for a sample application, provided in the Appendix and described in detail in Section 6.

The source code for an application written in the SPADE language is organized in terms of 5 main sections:

- **Application meta-information**: This section lists the application name, followed optionally by the debug/tracing level desired.
- **Type definitions**: This is where application designers must create a namespace for the types to be used by the application as well as, optionally, aliases to the types they intend to use. The type namespace provides type system-level isolation amongst System S applications that may be concurrently running on the system.
- **External libraries**: This is an optional section where application designers can include references to libraries and their file system paths, as well as the header files with interfaces for the external libraries employed by user-defined operators.
- **Node pools**: This is an optional section where pools of compute nodes can be defined. While an application written in SPADE can be fully optimized by the compiler (at compile-time) and by the System S resource management infrastructure and scheduler (at run-time), the creation of node pools provides a great deal of fine-level control over placement and partitioning during development and hand-optimization phases.
- **Program body**: This is where the application itself is described. SPADE's application description is stream-centric in the sense that streams are first class objects. The flow of computation is completely described by the streams an application produces.

A typical application will ingest an external (non-System S) data stream, creating a SPADE stream, process that stream through the utilization of one or more language-supported operators or user-defined operators, and, finally, externalize a data stream by producing a resulting flow that can be tapped by software components that are external to the System S infrastructure. Streams are created either by manipulating and converting data coming from an external source into a data flow understood by System S (using SPADÉs Source operator) or by performing a data transformation on an incoming stream, carried out by another SPADE language operator or user-defined operator (udop, for short). Once a SPADE stream is available, it can be externalized by creating sinks, whereby the flow of data is sent to entities that are outside of System S. The Sink operator used to perform this externalization can write to files, sockets, among other edge adapters.

SPADE supports having feedback loops within data flow-graphs, where a stream generated by a downstream operator is connected back into the input of an upstream operator. This is particularly useful for user-defined operators (see Section 5.1).

3.3 Operators

SPADE was conceived around the idea of a toolkit of operators. Currently, a single toolkit is available and it provides a collection of stream-relational operators. These operators can be used to implement any relational query with windowing extensions used in streaming applications. We are in the process of providing support for the creation of toolkits geared towards other application domains, such as signal processing and stream data mining. In lieu of these additional toolkits and as a means to allowing the creation of customized operators, the language supports the definition of user-defined operators (see Section 3.5).

The operators currently supported are the following:

- **Functor**: A Functor operator is used for performing tuple-level manipulations such as filtering, projection, mapping, attribute creation and transformation. In these manipulations, the Functor operator can access tuples that have appeared earlier in the input stream.
- **Aggregate**: An Aggregate operator is used for grouping and summarization of incoming tuples. This operator supports a large number of grouping mechanisms and summarization functions.
- **Join**: A Join operation is used for correlating two streams. Streams can be paired up in several ways and the join predicate, i.e., the expression determining when tuples from the two streams are joined, can be arbitrarily complex.
- **Sort**: A Sort operator is used for imposing an order on incoming tuples in a stream. The ordering algorithm can be tweaked in several ways.
- **Barrier**: A Barrier operator is used as a synchronization point. It consumes tuples from multiple streams, outputting a tuple only when a tuple from each of the input streams has arrived.
- **Punctor**: A Punctor operator is used for performing tuple-level manipulations where conditions on the current tuple as well as on past tuples are evaluated for generating punctuations[2] in the output stream.
- **Split**: A Split operator is used for routing incoming tuples to different output streams based on a user-supplied routing condition.

[2]Punctuations are out-of-band signals that mark window boundaries for operations that may rely on user-defined windows (e.g., sort, aggregate, and join).

- **Delay**: A Delay operator is used for delaying a stream based on a user-supplied time interval.

3.4 Edge Adapters

Edge adapters in the SPADE language are also described as language operators – source and sink:

- **Source**: A Source operator is used for creating a stream from data flowing from an external source. This operator is capable of performing parsing and tuple creation, and can interact with a diverse set of external devices.

- **Sink**: A Sink operator is used for converting a stream into a flow of tuples that can be used by components that are not part of System S. Its main task consists of converting tuples into objects accessible externally through devices such as the file system or the network.

The external resources referred to by the edge adapters are specified by a URI (Universal Resource Locator). The URI information is used by the code generator to appropriately select external libraries as well as other resource-specific configurations. In the cases we have seen so far, the authentication and configuration issues associated with accessing external resources can be dealt with by crafting URIs with the necessary information or, in some cases, having the URI refer to a configuration file with additional information, including communication protocols, message formats, among other things.

3.5 User-Defined Operators

The SPADE language provides the capability for extending the basic building block operators by supporting user-defined operators (udops). These operators can make use of external libraries and implement operations that are customized to a particular application domain. For example, suppose a package for performing data mining on streams is available. The udop support enables an application to receive tuples from SPADE streams, hand them over to the external package, perform computations, and, eventually, originate SPADE streams for downstream consumption.

The udop code generator creates skeleton code and seamlessly integrates the user-defined operator dependencies in the overall build process. From the skeleton code, the application developer can tap the resources of the external libraries, customizing the internal operator processing to the needs of her application. Currently, the skeleton code is generated in C++, which allows for the easy integration of existing analytics, speeding up the process of integrating legacy code. As will be seen in Section 4, user-defined operators are also targeted by the the optimizer. In other words, built-in and user-defined operators alike are seamlessly processed by the code optimizer in building the corresponding System S application.

In our experience, we have seen developers employing udops for a wide-range of reasons: From converting legacy applications to System S so that they can run in a stream environment, to wrapping external stream data mining libraries (such as VFML [14]), to interfacing with external platforms (such as extracting performance metrics from IBM DB2 [9]), among others.

3.6 Advanced Features

As we have previously mentioned, System S was conceived to support a wide-range of stream processing applications. The implication to the SPADE programming language is the need to support a richer set of features than typically found in other stream processing platforms. These features span not only what have been made available by the stream engines developed by other groups, but also those aimed at providing language constructs and mechanisms to simplify the construction of System S applications. Three advanced features that we deem particularly important are discussed in this section.

3.6.1 List Types and Vectorized Operations

The SPADE language includes native supports for list types as well as vectorized operations on them. In the domains of signal processing, data mining and pattern classification, straightforward vector manipulation is fundamental in approximating the problem formulation to its representation in terms of source code. Thus, such an approach improves usability. In SPADE attributes with list types are created in one of the following three ways: (1) by reading them from an external source via the Source operator, (2) by using a Functor operator to create a new list attribute, or (3) by using an Aggregate operator to *collect* attributes from multiple tuples into a single list. SPADE supports expressions that mix and match list and scalar types. Moreover, many of the built-in SPADE functions are list-type capable, that is they accept list types where scalars are expected, and produce a list result accordingly. List types also enable easy string manipulations using regular expression matching, where match results are stored in a string list.

3.6.2 Flexible Windowing Schemes

Another important language feature is the support for sophisticated *windowing* mechanisms. Several of the SPADE's built-in operators are configurable in terms of their windowing behavior. Generally, there is support for *tumbling* windows, *sliding* windows, and *punctuation-based* windows. Sophisticated combinations of windowing boundaries and slide factors are possible (e.g., count, time, and attribute-based windows, slides, and their *combinations*), but a complete discussion is beyond the scope of this paper. Instead, we will next sketch some of the fundamental windowing characteristics. The most simple windowing schema is tumbling windows. Tumbling windows are operated on and then flushed when they become full (e.g., after a fixed number of tuples have been accumulated). Sliding windows on the other hand have two components: an expiration policy and a trigger mechanism. The expiration policy defines when accumulated tuples are purged and, therefore, they are no longer part of the internal state carried by an operator (e.g., last 100 most recent tuples are kept around). The trigger mechanism flags when the aggregation operation should take place (e.g., an aggregation should be made and output every time a new tuple is received by the operator). Finally, the support for punctuation-based windows allows the creation of explicit window boundaries in a stream. Such boundaries can be created by user-defined operators as well as by certain built-in operators. For a punctuation-based window, an operator accumulates tuples until a punctuation is received. Once the punctuation is received, the operation is performed. Several rules govern how punctuations are propagated through a query network. A complete discussion of these rules is beyond the scope of this paper, however.

3.6.3 *Pergroup Aggregates and Joins*

Finally, a novel aspect in SPADE is the support for a *grouping* mechanism, associated with tumbling windows as well as sliding windows. The grouping mechanism is currently a feature associated with the aggregate and equi-join operators. It determines the *scope* for the expiration and trigger policies. In essence, they can apply to the whole window or to every distinct group currently in the window. For example, if an operator is configured such that it must keep 200 tuples, the SPADE language has a notation to specify whether 200 tuples should be aggregated (or joined) for the whole window or for each group that it might be aggregating (or joining). This approach allows the simultaneous and independent aggregation/joining for distinct groups, which for a large number of groups typically translates into substantial computational savings [3].

3.7 Application Interoperability

As we have discussed, System S provides a wide-spectrum landscape for application development. In such an environment, an important aspect is application interoperability. The basic wiring mechanism provided by the Stream Processing Core is fairly malleable – i.e., processing elements can be connected to each other by hardwiring a connection or dynamically, by having a processing element specify a subscription flow specification expression, which determines the properties of streams to be consumed.

A SPADE application can, in a controllable fashion, interoperate with other SPADE applications as well as with any other System S application at runtime. While an application stream is, by default, only available to other consumer operators in the application that defines it, a stream can be made visible by *exporting* it. Conversely, external streams can be consumed by employing a reference to a *virtual* stream.

4. COMPILER OPTIMIZATIONS

In this section we discuss three optimization opportunities created by SPADE's code generation framework. The first one is the *operator grouping* optimization, which deals with the mapping of operators into PEs. The second one is the *execution model* optimization, which deals with the assignment of threads to operators. Finally, the third one is the *vectorized processing* optimization, which deals with the hardware acceleration of vectorized operations on lists.

4.1 Operator Grouping Optimization

The System S runtime deals with the scheduling and placement of PEs, whereas it does not handle operators directly. It is the responsibility of the SPADE compiler to map operators into PEs. The naïve approach of mapping each operator to a different PE results in significant overheads due to several factors. First, each PE is a separate execution unit and thus having as many PEs as operators implies executing large number of processes or threads. This will limit the performance, especially when the job is not distributed over a large number of nodes, resulting in higher number of threads per node. Second, streaming data items from one PE to another involves message transmission and queuing delays, and thus having too many PEs will cause additional delays at each link of the data-flow graph. This will increase latency especially when the job is distributed over a large number of nodes (higher transmission delays). On the other hand, the other extreme case of having a single PE obviously prevents us from making use of processing power from multiple nodes. Therefore, the goal of the operator grouping optimization is to find the best balance between these two extremes.

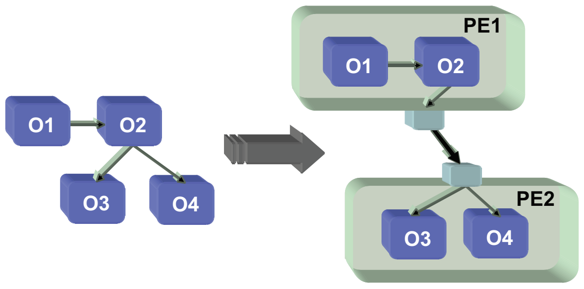

Figure 4: Example operator to PE mapping

Figure 4 shows an example operator-to-PE mapping, where four operators are mapped into two PEs. Note that PEs have buffers attached to their input/output ports. When PEs are located in different nodes, tuples are marshaled into SDOs and transferred over the network from input buffers to output buffers. In contrast, only a pointer is passed around when the PEs are co-located on the same node and sit under the same PE container. The intra-PE transfers among operators within the same PE are significantly more efficient than their inter-PE counterparts. This will become clearer in the next section, where we describe SPADE's *optimizing partitioner* used for operator-to-PE mapping.

4.2 Execution Model Optimization

As we discussed, SPADE can map a set of operators into a PE. By default, operators that are part of a *composite* PE do not run parallel threads. Optionally, SPADE can assign multiple parallel threads to operators. With the current trend of increasing the number of cores in CPU hardware to improve performance, multi-threading becomes an important aspect of high-performance applications. Making use of multiple cores at the level of operators entails generating multi-threaded code for the built-in operators. For instance, an Aggregate operator can potentially use multiple threads to compute aggregates defined over different attributes, in parallel. Assuming built-in operators have parallel implementations, the high-level problem is to decide how to best distribute threads to operators within a PE. At the time of this writing, SPADE does not generate parallel code for individual operators. To make use of multiple cores on a single node, SPADE creates multiple PE's to be run on the same node. We are currently working on code generation for multi-threaded built-in operators and the problem of optimizing the thread-to-operator mapping.

4.2.1 *Alternative Hardware Architectures*

New opportunities in optimizing the execution model arise with the increasing diversity of the hardware available for general purpose computing, such as the Cell processor [15], graphics processors (GPUs) [18], network processors [16], etc. Acceleration of data stream operators on these hardware often require a different execution model and specialized code. We have performed prototype implementations of stream joins (see [11]) and sorting (see [10]) on the Cell processor. Integrating full Cell support into SPADE will involve developing code generators to specialize these implementations for given operator configurations[3].

4.3 Vectorized Processing Optimization

The vectorized operations on list types can be accelerated through Single-Instruction Multiple-Data (SIMD) operations available in most modern processors. SPADE utilizes Streaming SIMD Extensions (SSE) on the Intel processors to accelerate the basic arithmetic operations on list types.

5. SPADE'S OPTIMIZING PARTITIONER

SPADE's optimizing partitioner uses *operator fusion* as the underlying technique for forming PEs out of operators and employs a two-phase learning-based optimization approach to configure the operator partitions.

5.1 Operator Fusion

SPADE uses code generation to fuse operators into PEs. Concretely, a PE generator produces code that (1) fetches tuples from the PE input buffers and relays them to the operators within, (2) receives tuples from operators within and inserts them into the PE output buffers, and (3) for all the intra-PE connections between the operators, it *fuses* the outputs of operators with the inputs of downstream ones using *function calls*. This fusion of operators with function calls results in a depth-first traversal of the operator subgraph that corresponds to the partition associated with the PE, with no queuing involved in-between.

As noted earlier, SPADE supports multi-threaded operators, in which case the depth-first traversal performed by the main PE thread can be cut short in certain branches, where separate threads can continue from those branches independently. The latter requires operators to be thread-safe. For user-defined operators, SPADE automatically protects the process methods to provide thread-safety. For built-in operators, finer grained locking mechanisms are used for this purpose. SPADE code generators do not insert these locks into the code if an operator is not grouped together with other operators and is part of a *singleton* PE.

Since SPADE supports feedback loops in the data-flow graph, an operator graph is not necessarily cycle-free, which opens the possibility of infinite looping within a composite PE. The rationale behind allowing feedback loops in SPADE is to enable udops to *tune* their logic based on feedback from downstream operators. Under operator fusion, SPADE does not allow feedback links into built-in operators and expects feedback links into udops to be connected to non-tuple-generating inputs. This guarantees cycle free operation under operator fusion. In our experience, feedback loops have been a valuable asset in developing applications that are heavy on udops, an example of which is a semiconductor fabrication line monitoring application (built with SPADE) that uses downstream yield statistics to tune its upstream detection algorithm.

[3]The System S runtime is already ported to the PowerPC architecture. A PE can run on the PPE side of a Cell processor, and can interact with the SPE threads to accelerate processing of data streams.

5.2 Statistics Collection

In order to decide on how to best partition the operators into PEs, SPADE needs to know resource usage characteristics of operators. Such characteristics are dependent on the workload, as well as the specific configurations of the operators. Even though the internal mechanics of built-in SPADE operators are known and thus a cost model can potentially be build, the same does not hold for user-defined operators. Moreover, heavy use of functions within expressions that appear in built-in operators makes it harder to come up with accurate cost models. Relying on the long-running nature of SPADE jobs, we adopted a learning-based statistics collection framework. Before compiling a SPADE job for the final execution, we compile it in a special statistics collection mode. The application is then run in this mode to collect runtime information for each operator and each link in the data-flow graph. These statistics include metrics such as CPU load and network traffic. After this information is collected, the application is compiled for a second time. In this second compilation step, the SPADE optimizer uses the statistics collected in the earlier step to come up with an optimized operator grouping, and applies operator fusion to yield the composite PEs. At this point the long-running SPADE job is ready to be deployed.

5.3 Optimization Goal

Given the CPU load and network traffic statistics for the data-flow graph at hand, SPADE's optimizing partitioner aims at minimizing the total inter-PE communication, while respecting the constraint that the total load imposed by the operators within a PE should not exceed the capacity of a single processor. The optimizer will pick the smallest number of nodes that satisfy this constraint. Even though this strategy is more tailored towards throughput optimization, it also works well for reducing the latency. Our experience has shown that crossing PE boundaries is a major cause of increased latency. Thus, the conservative nature of SPADE's optimizing partitioner with respect to creating additional PEs reduces the latency in general.

6. AN EXAMPLE SPADE APPLICATION

In this section we describe a sample SPADE application from the finance engineering domain, called *Bargain Index Computation*. We also present a performance study of a parallel and distributed version of this application, to give the reader an idea of the level of performance and scalability achievable with SPADE and System S.

6.1 Bargain Index Computation

We consider a stock trading scenario, where the aim is to find *bargains* to buy. A sell *quote* for a given stock is considered a bargain, if it is available in quantity and at a cheaper price relative to its moving average price as seen in recent *trades*. Bargain index is a scalar value representing the magnitude of the bargain, i.e. how much of a bargain it really is. We would like to compute the bargain index for every stock symbol that appears in the source stream. The visual representation of the SPADE query that implements this logic using built-in SPADE operators is given in Figure 5. In what follows, we describe the source data, the processing logic, and the result management aspects of this application.

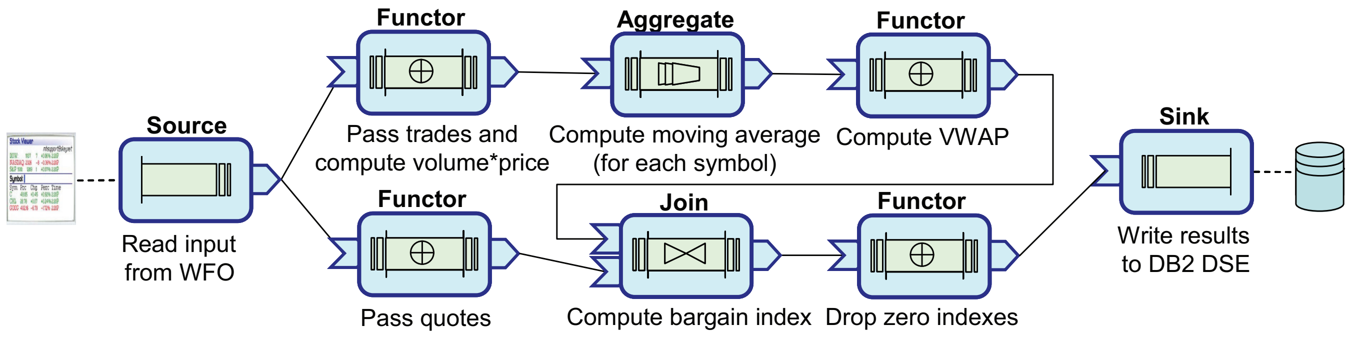

Figure 5: Bargain Index computation for *all* stock symbols

Source Data: Trades and Quotes

The source data contains trade and quote information. A trade shows the price of a stock that was just traded, whereas a quote reveals either an "ask" price to sell a stock or a "bid" price to buy a stock. For this application, we only consider the ask price in the quote data, i.e., we care about sell quotes and ignore the buy ones. Table 1 lists relevant fields of the trade and quote data. Note that, each trade has an associated price and a volume, whereas each quote has an associated ask price and an ask volume. In Figure 5, live stock data is read directly from the IBM WebSphere Front Office (WFO) [24] – a commercial middleware platform for performing front office processing tasks in financial markets. SPADE's source operator has built-in support for tapping into WFO sources and converting them into SPADE streams.

Ticker	Type	Price	Volume	Ask Price	Ask Size
MWG	Trade	24.27	500	–	–
TEO	Quote	–	–	12.85	1
UIS	Quote	–	–	5.85	6
NP	Trade	28.00	5700	–	–
TEO	Trade	12.79	700	–	–

Table 1: Sample trade and quote data (relevant fields shown)

Processing Logic: Bargain Detection

To compute the bargain index, we first need to separate the source stream into two branches, trades and quotes. This is achieved via the use of two Functor operators (see Figure 5). The Functor operator that creates the upper trade branch also computes *trade price* × *volume*, which will later be used to compute the *volume weighted average price* (VWAP) – a commonly used metric in algorithmic trading. The Aggregate operator that follows the Functor computes a moving sum over *price* × *volume* and volume. It uses a *per-group* window of size 15 tuples with a slide of 1 tuple, i.e. it outputs a new aggregate every time it receives a trade, where the aggregate is computed over the last 15 tuples that contained the same stock symbol of the last received trade. Another Functor operator is used to divide the moving summation of price × volume to that of volume, giving the most recent VWAP value for the stock symbol of the last received trade. The resulting intermediate stream is connected to the first input of an equi-Join (on stock symbol) operator, which has a *per-group* window of size 1 tuple on the same input. In other words, the join window for the first input has one group for each unique stock symbol seen so far and stores the last VWAP value within each group. The second input of the join is connected to the quote stream, and has a zero-sized window (this is a single-sided join). The aim is to associate the last received quote with the most recent VWAP value computed for the stock symbol of that quote. Once this is done, a simple formula is used to compute the bargain index as a function of the ask price, ask size, and the VWAP value. A final Functor operator filters out the bargain indexes that are zero, indicating that a bargain has not been detected.

Results Management and DB2 DSE

In Figure 5, the non-zero bargain index values are fed into a Sink operator, which is connected to IBM DB2 Data Stream Edition [9] – an extension of DB2 designed for persisting high-rate data streams. The result database can potentially be connected to an automated trading platform, in order to act upon the bargain index results.

6.2 A Parallel Version for Historical Data

We now present a parallel and distributed version of the same query and provide brief performance results. To showcase scalability, we use historic market feed data stored on the disk. The data set contains 22 days' worth of ticker data (the month of December 2005) for ≈ 3000 stocks with a total of ≈ 250 million trade and quote transactions, resulting in ≈ 20GBs of data. It is organized as one file per day on the disk, and is stored on IBM's General Parallel File System (GPFS) [13] – a commercial high-performance shared-disk clustered file system. For this workload, we run the bargain index computation query and store the detected bargains back into output files on GPFS.

To parallelize the processing we run 22 copies of the flow depicted in Figure 5, one for each trading day. This is achieved using a SPADE *for loop* construct that encloses the complete query specification. For performance reasons, we run operators that are part of the processing of the same day within a single PE. We distribute these PEs over 16 nodes in our cluster. SPADE is capable of expressing more sophisticated parallelization and distribution schemes, yet for this application a simple coarse-grained scheme is sufficiently effective. The SPADE code for this parallel version of the application is given in the Appendix.

It is important to note that a developer interacts only with the SPADE language and the compiler, for generating a par-

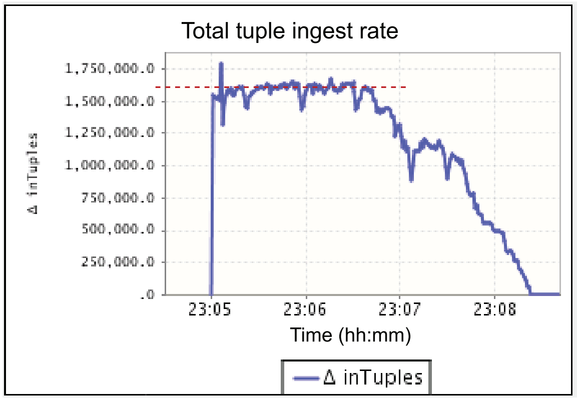

Figure 6: Tuple ingestion rate for the parallel and distributed bargain index computation application, using 22 parallel queries distributed over 16 nodes.

allel and distributed System S application, like the bargain index computation one we have described in this section. The resulting application runnables are easily deployable on the System S cluster using convenience scripts automatically generated by the SPADE compiler, which in turn rely on the System S job management infrastructure. Figure 6 shows the performance obtained from running our example application. The figure plots the aggregate tuple ingestion rate as a function of the current wallclock time. Note that the sustained processing rate is around 1.6 million tuples/sec and the total time required to compute all the bargain index values for a month's worth of disk resident data takes less than 3.5 minutes. The downward trend in the aggregate ingestion rate after the initial flat plateau is due to some of the daily subqueries completing earlier than some others, since different days have differing trading volumes. Moreover, recall that 22 queries are distributed over 16 machines, which results in further imbalance in the server loads. This is because all operators within the same query are packed into a single PE, resulting in 22 units that are distributable over 16 machines. However, our choice of 16 nodes for this experimental study was not arbitrary. We picked the available nodes that have high-bandwidth access to the GPFS file system, in order to avoid a potential file system bottleneck.

7. ONGOING AND FUTURE WORK

The System S platform in general, and SPADE in particular are both under very active development at IBM Research. While several real-world applications have been implemented using SPADE demonstrating its usability, many new improvement fronts have also been opened.

An important ongoing development is the implementation of a rapid-application development environment based on visual metaphors for composing an application. With such an environment, developers will be able to compose stream flows visually, but as importantly, they will be able to control parallelization and distribution characteristics, manually tweak optimization choices, debug applications, visualize performance metrics, among other tasks. Performing several of these tasks is already possible, however, the ability to do them in an integrated and visual manner will make the application development and improvement cycle much easier.

The addition of new and domain-specific operator toolkits is also ongoing. We are in the process of identifying and selecting the important building blocks commonly used in the domains of signal processing, stream data mining, and financial engineering with the aim of making them full-fledged built-in operators in the language.

On the code generation side, we are actively working on integrating special-purpose hardware platforms with the goal of providing unprecedented performance to time-critical stream processing applications.

At the other end of the spectrum, we are also working towards front-ending SPADE with higher-level languages including Stream SQL and System S's semantic composition framework [19], allowing these different front-ends to automatically and efficiently output System S artifacts, leveraging System S capabilities to the fullest extent.

Finally, we are improving the interoperability features of SPADE. On the one hand we are including additional edge adapters for both data ingestion and data externalization, spanning from support for binary data and HTTP-based protocols to the integration with other middleware platforms. On the other hand, we are increasing the capabilities of SPADE applications to interact with non-SPADE System S applications, leveraging System S' capabilities for on-the-fly dynamic application composition [17].

8. CONCLUSIONS

In this paper we have presented System S' SPADE – a declarative stream processing engine. We believe that SPADE is introducing a programming framework that is novel in many ways compared with the existing stream processing middleware platforms. Among its novel features, we believe that the native support for edge adapters and toolkits of operators as well as the reliance on code generation, coupled with the optimization framework make SPADE particularly suitable for building high-performance scalable stream processing applications. Also its ability to extend the collection of edge adapters and toolkits make it adequate for developing applications geared towards application domains we have not yet even contemplated. While SPADE is labeled as an intermediate language, it is directly usable by application developers for quickly prototyping complex applications. Nevertheless, the advances discussed in Section 7 will further decrease the barrier of entry for novice developers. We have also shown performance results for a relatively complex stream processing application built completely with SPADE lending quantitative credibility to our optimization strategy as well as System S' high-performance capabilities. In conclusion, despite the many challenges ahead, we believe that SPADE already provides an interesting and important set of abstractions for composing and building large-scale, distributed, and scalable stream processing applications.

Acknowledgments

We would like to acknowledge Nagui Halim, the principal investigator of the System S project for providing us with invaluable guidance throughout the development of SPADE. We thank Olivier Verscheure and Deepak Turaga for being the early adopters of SPADE and using it for the implementation of real-world System S pilots. We thank Senthil Nathan for developing a comprehensive collection of sample applications that illustrates different aspects and capabilities of

SPADE as well as his help with the integration of IBM DSE. We thank Philippe Selo for his help with the IBM WFO integration as well as general System S issues. Finally, we acknowledge Kevin Pleiter and Chitra Venkatramani for participating in several discussions and providing guidance on application scenarios in financial markets.

9. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the Conference on Innovative Data Systems Research, CIDR*, 2005.

[2] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. SPC: A distributed, scalable platform for data mining. In *Proceedings of the Workshop on Data Mining Standards, Services and Platforms, DM-SSP*, 2006.

[3] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu. On optimizing aggregations and joins for high-performance data stream processing. In *to be submitted to International Conference on Supercomputing, ACM ICS*, 2008.

[4] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The Stanford stream data manager. *IEEE Data Engineering Bulletin*, 26, 2003.

[5] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. Technical report, InfoLab – Stanford University, October 2003.

[6] H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on Aurora. *VLDB Journal, Special Issue on Data Stream Processing*, 2004.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the Conference on Innovative Data Systems Research, CIDR*, 2003.

[8] Coral8, inc. http://www.coral8.com/, May 2007.

[9] IBM DB2. http://www.ibm.com/db2, October 2007.

[10] B. Gedik, R. R. Bordawekar, and P. S. Yu. CellSort: High performance sorting on the Cell processor. In *Proceedings of the Very Large Data Bases Conference, VLDB*, 2007.

[11] B. Gedik, P. S. Yu, and R. R. Bordawekar. Executing stream joins on the Cell processor. In *Proceedings of the Very Large Data Bases Conference, VLDB*, 2007.

[12] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden. XStream: A signal-oriented data stream management system. In *Proceedings of the International Conference on Data Engineering, IEEE ICDE*, 2008.

[13] IBM general parallel file system. http://www.ibm.com/systems/clusters/software/gpfs, October 2007.

[14] G. Hulten and P. Domingos. VFML – a toolkit for mining high-speed time-changing data streams. http://www.cs.washington.edu/dm/vfml/, 2003.

[15] IBM. Cell Broadband Engine architecture. Technical Report Version 1.0, IBM Systems and Technology Group, 2005.

[16] Intel. IXP2400 network processor hardware reference manual. Technical report, Intel Corporation, May 2003.

[17] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *Proceedings of the International Conference on Management of Data, ACM SIGMOD*, 2006.

[18] P. Kipfer and R. Westermann. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 46. Addison Wesley, 2005.

[19] Z. Liu, A. Ranganathan, and A. V. Riabov. Use of OWL for describing stream processing components to enable automatic composition. In *OWL: Experiences and Directions, OWLED*, 2007.

[20] Mathworks MATLAB. http://www.mathworks.com/, October 2007.

[21] StreamBase Systems. http://www.streambase.com/, May 2007.

[22] IBM UIMA. http://www.research.ibm.com/UIMA/, Aug 2007.

[23] J. D. Ullman. *Database and Knowledge-Base Systems*. Computer Science Press, 1988.

[24] IBM WebSphere front office for financial markets. http://www.ibm.com/software/integration/wfo, October 2007.

[25] K.-L. Wu, P. S. Yu, B. Gedik, K. W. Hildrum, C. C. Aggarwal, E. Bouillet, W. Fan, D. A. George, X. Gu, G. Luo, and H. Wang. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S. In *Proceedings of the Very Large Data Bases Conference, VLDB*, 2007.

APPENDIX: Spade source code

```
# %1 and %2 are the first and second parameters
#define NCNT min(%1,16)  #* number of nodes to utilize *#
#define FCNT min(%2,30)  #* number of days to analyze  *#

[Application]
vwap # trace

[Typedefs]
typespace vwap

[Nodepools]
nodepool ComputingPool[16] := () # automatically allocated from available nodes

[Program]
#* Source data format:
 * 1 ticker:String,       8 volume:Float,      15 askprice:Float,    22 peratio:Float,
 * 2 date:String,         9 vwap:Float,        16 asksize:Float,     23 yield:String,
 * 3 time:String,         10 buyer:String,     17 nsellers:Float,    24 newprice:Float,
 * 4 gmtoffset:Integer,   11 bidprice:Float,   18 qualifiers:String, 25 newvolume:Float,
 * 5 ttype:String,        12 bidsize:Float,    19 seqn:Long,         26 newseqn:Long,
 * 6 ex:String,           13 nbuyers:Float,    20 exchtime:String,   27 bidimpvol:String,
 * 7 price:Float,         14 seller:String,    21 blocktrd:String,   28 askimpvol:String,
                                                                     29 impvol:String     *#
for_begin @day 1 to FCNT # for each day

  stream TradeQuote@day(ticker:String, ttype:String, price:Float, volume:Float, askprice:Float, asksize:Float)
      := Source()["file:////gpfs/ss/taq_200512"+select(@day<10,"0@day","@day")+".csv", nodelays, csvformat]
           { 1, 5, 7-8, 15-16 }
    -> partition["mypartition_@day"], ComputingPool[mod(@day-1,NCNT)]

  stream TradeFilter@day(ticker: String, myvwap:Float, volume:Float)
      := Functor(TradeQuote@day) [ttype="Trade" & volume>0.0]
           { myvwap := price*volume }
    -> partitionFor(TradeQuote@day), ComputingPool[mod(@day-1,NCNT)]

  stream QuoteFilter@day(ticker:String, askprice:Float, asksize:Float)
      := Functor(TradeQuote@day) [ttype="Quote" & askprice>0.0]{}
    -> partitionFor(TradeQuote@day), ComputingPool[mod(@day-1,NCNT)]

  stream VWAPAggregator@day(ticker:String, svwap:Float, svolume:Float)
      := Aggregate(TradeFilter@day <count(15), count(1), pergroup>) [ticker]
           { Any(ticker), Sum(myvwap), Sum(volume) }
    -> partitionFor(TradeQuote@day), ComputingPool[mod(@day-1,NCNT)]

  stream VWAP@day(ticker:String, cvwap:Float)
      := Functor(VWAPAggregator@day) [true]
           { cvwap := (svwap/svolume)*100.0 }
    -> partitionFor(TradeQuote@day), ComputingPool[mod(@day-1,NCNT)]

  stream BargainIndex@day(ticker:String, bargainindex:Float)
      := Join(VWAP@day <count(1), pergroup>; QuoteFilter@day <count(0)>)
           [{ticker}={ticker}, cvwap > askprice*100.0]
           { bargainindex := exp(cvwap-askprice*100.0)*asksize }
    -> partitionFor(TradeQuote@day), ComputingPool[mod(@day-1,NCNT)]

  export stream NonZeroBargainIndex@day(schemaof(BargainIndex@day))
      := Functor(BargainIndex@day) [bargainindex>0.0] {}
    -> partitionFor(TradeQuote@day), ComputingPool[mod(@day-1,NCNT)]

  Null := Sink(NonZeroBargainIndex@day) ["file:///Bargains@day.dat"]{}
    -> partitionOf(TradeQuote@day), ComputingPool[mod(@day-1,NCNT)]

for_end
```